Big-oh → upper bounds

T(n) is O(f(n)) if there exist constants $c>0$ and $n_0 \ge 0$ such that $T(n) \le c \cdot f(n)$ for all $n \ge n_0$.

Big-omega → lower bounds

T(n) is $\Omega(f(n))$ if there exist constants $c>0$ & $n_0 \ge 0$ such that $T(n) \ge c \cdot f(n)$ for all $n \ge n_0$.

Big-Theta → Tight bounds $c_1>0$, $c_2>0$, $n_0 \ge 0$

$c_1 \cdot f(n) \le T(n) \le c_2 \cdot f(n)$ for all $n \ge n_0$

Little-o – T(n) is o(f(n)) if for any constant $c>0$ there exists $n_0 \ge 0$ such that $T(n) < c \cdot f(n)$ for all $n \ge n_0$.

Little-w: $T(n) > c \cdot f(n)$

o( ) is like <   O( ) is like ≤   Θ( ) is like =

w( ) is like >   Ω( ) is like ≥

o → $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ $\lg n = \log_2 n$

w → $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ $\ln n = \log_e n$

Arithmetic series $\sum_{k=1}^{n} k = 1+2+\dots+n = \frac{n(n+1)}{2}$

Geometric series $\sum_{k=0}^{n} x^k = 1+x+\dots+x^n = \frac{x^{n+1}-1}{x-1}$ $(x \ne 1)$

Special case $|x|<1$: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

Harmonic series $\sum_{k=1}^{n} \frac{1}{k} = 1+\frac{1}{2}+\dots+\frac{1}{n} = \log_e n$

$\sum_{k=1}^{n} \log k \approx n\log n - n$

$\sum_{k=1}^{n} k^P = 1^P + 2^P + \dots + n^P \approx \frac{1}{P+1} n^{P+1}$

Mathematical Induction:-

Basis step:- prove the statement is true for base case

Inductive step:- assume that statement is true for n (or all integers $\le n$) and then prove that statement is true for n+1

$T(n) = T(n-1) + n$ $O(n^2)$

$T(n) = T(n/2) + c$ $O(\log n)$

$T(n) = T(n/2) + n$ $O(n)$

$T(n) = 2T(n/2) + 1$ $O(n)$

Substitution method:-

Guess the solution works

use induction to prove that the solution

Ex- $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Guess $T(n) = O(n \lg n)$ need to prove $T(n) \le cn\lg n$

True for $m<n$

$m = \lfloor n/2 \rfloor$

$T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor$

Substitute in main eqn

$\Rightarrow T(n) = 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n$

$T(n) \le cn\lg(n/2) + n$

$= cn\lg n - cn\lg 2 + n$

$= cn\lg n - cn + 1$

$\le cn\lg n$

Recursion tree method:- convert into a tree each node represents the cost incurred at various levels of recursion.

Sum up all the costs of all levels.

Ex $T(n) = T(n/2) + T(n/3) + n$

h = $\log_2 n$ h = $\log_3 n$

n

n/2 n/3

n/4 n/6 n/6 n/9

1 2/3 1 4/3 (empty)

→ n

→ $\frac{5n}{6}$

→ $(\frac{5}{6})^2 n$

→ $c_1 \le T(n) \le c_2$

$\Rightarrow n + \frac{5}{6}n + (\frac{5}{6})^2 n + \dots + (\frac{5}{6})^{\log_2 n} n \le T(n) \le$

$n + \frac{5}{6}n + (\frac{5}{6})^2 n + \dots + (\frac{5}{6})^{\log_3 n} n$

$\Rightarrow O(n) \le T(n) \le O(n) \Rightarrow \Theta(n)$

Master's Theorem:-

$T(n) = aT(n/b) + f(n)$

where $a \ge 1$, $b > 1$ and $f(n) > 0$

case

1: If $f(n) = O(n^{\log_b^a - \varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b^a})$ εd

2: If $f(n) = \Theta(n^{\log_b^a})$ then

$T(n) = \Theta(n^{\log_b^a} \log n)$ $\varepsilon = d$

3: If $f(n) = \Omega(n^{\log_b^a + \varepsilon})$ for some $\varepsilon > 0$ and if $af(n/b) \le cf(n)$ for some $c<1$ and all sufficiently large n, εd

then $T(n) = \Theta(f(n))$.

Divide & Conquer

Divide the problem into a number of sub problems of smaller size

Conquer the subproblems recursively

Combine the solutions of subproblems

Maximum Subarray:-

A[1...n] → A[1...x] → sum maximum

Find max cross subarray (A, low, mid, high)

left sum = −∞

sum = 0

for i = mid downto low

sum = sum + A[i]

if sum > left-sum then

leftsum = sum

max left = i

right sum = −∞

→ sum = 0

for j = mid + 1 to high

sum = sum + A[j]

if sum > right-sum then

right-sum = sum

max right = j

return (maxleft, max right, leftsum + rightsum)

total time $T(n) = 2T(n/2) + \Theta(n)$

$\Rightarrow T(n) = O(n\log n)$

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < 2^n < n!$

Insertion sort

5 2 4 6 1 3 → 2 5 4 6 1 3

→ 2 4 5 6 1 3 → 2 4 5 6 1 3

→ 1 2 4 5 6 3 → 1 2 3 4 5 6

Insertion sort (A)

for j = 2 to A.length

key = A[j]

// Insert A[j] into sorted sequence A[1..j−1]

i = j − 1

while i > 0 & A[i] > key

A[i+1] = A[i]

i = i − 1

A[i+1] = key.

almost sorted arrays O(n)

$O(n^2)$ running time in worst and average case

Bubble sort → repeatedly swaps through the array

Swap adjacent element that are out of order

Sorts in place → Bubble, Insertion

Running time $O(n^2)$

Easier to implement but slower than insertion sort.

Merge sort A[P, r]

Divide → Conquer → Combine

5 2 4 7 1 3 2 6   1 2 2 3 4 5 6 7

5 2 4 7   1 3 2 6   2 4 5 7   1 2 3 6

5 2   4 7   1 3   2 6   2 5   4 7   1 3   2 6

5 2 4 7 1 3 2 6   5 2 4 7 1 3 2 6

Running time

Divide → compute q → $D(n) = \Theta(1)$

Conquer → solve 2 subproblems → $2T(n/2)$

Combine → $\Theta(n)$

$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$

⇒ case 2 master theorem

$T(n) = O(n\lg n)$

Algorithm

Merge (A, P, q, r)

$n_1 = q - p + 1$

$n_2 = r - q$

let $L[1 \dots n_1+1]$ & $R[1 \dots n_2+1]$ be new arrays

for i = 1 to n_1

L[i] = A[P + i − 1]

for j = 1 to n_2

R[j] = A[q + j]

$L[n_1+1] = \infty$

$R[n_2+1] = \infty$

i = 1

j = 1

for k = p to r

if L[i] ≤ R[j]

A[k] = L[i]

i = i + 1

else A[k] = R[j]

j = j + 1

mergesort(A, P, r)

if p < r

q = ⌊(p+r)/2⌋

mergesort (A, P, q)

mergesort (A, q+1, r)

merge (A, P, q, r)

Bubble sort (A)

for i = 1 to A.length − 1

for j = A.length down to i+1

if A[j] < A[j−1]

exchange A[j] with A[j−1]

Selection sort:-

n = A.length

for i = 1 to n−1

minIndex = i

for j = i+1 to n

if A[j] < A[minIndex]

minIndex = j

Swap(A[i], A[minIndex])

Loop invariant of Selection sort

At the start of the loop in line 1 the subarray A[1...i−1] consists of the smallest i−1 elements in array A with sorted order

Proving Loop invariants

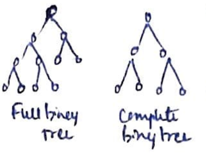
Initialization (Base case):- It is true prior to the first iteration of the loop

Maintenance (Inductive step): If it is true before an iteration of the loop, it remains true before the next iteration

Termination: when the loop terminates, the invariant gives us a useful property that helps show the algorithm is correct.

→ Stop the induction when the loop terminates

Tree :- degree → No of children
depth → Length of simple path from root to x
Level → all nodes at the same depth
height → Length of the longest simple path from x downward to some leaf node.



Full → each node is either a leaf or has exactly degree 2
complete → all internal nodes have degree 2

→ at most 2^l nodes at level l of binary tree
→ with depth d has at most $2^{d+1}-1$ nodes
⇒ with n nodes has depth at least $\lfloor \lg n \rfloor$

$$n \le \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$

Heap :→ complete binary tree with structural property : all levels are full, except possibly the last one, which is filled from left to right.
other (heap) property :- for any node x
parent (x) ≥ x.

Array Representation of Heaps :
Root A[1], Node is A[i], Left child of node i = A[2i], Right child of node i = A[2i+1]
Parent of node i = A[⌊i/2⌋]
elements in subarray A[(⌊n/2⌋+1)...n] are leaves
max heaps (largest element at root)
→ A[parent(i)] ≥ A[i]
min heaps (smallest element at root)
→ A[parent(i)] ≤ A[i]

Max Heapify (A, i)
{ l = left(i); r = Right(i);
 if (l <= heap-size(A) && A[l] > A[i])
  largest = l;
 else largest = i;
 if (r <= heap-size(A) && A[r] > A[largest])
  largest = r; $O(\lg n)$
 if (largest != i)
  Swap (A, i, largest);
  Heapify (A, largest);
};

Assumptions:- Left & Right subtrees of i are max heaps. A[i] may be smaller than its children.

Building a Heap → convert an array A[1...n] into a max heap (n = length[A])
→ elements in the subarray A[(⌊n/2⌋+1...n] are leaves.
→ Apply max Heapify on elements between 1 & ⌊n/2⌋

Build_max_Heap(A) $O(n)$
n = length (A)
for i ← ⌊n/2⌋ downto 1
 do Max heapify (A, i, n)

Heapsort :
→ Build a max heap from the array
→ Swap the root with the last element in the array
→ Discard this last node by decreasing the heapsize
→ call Max heapify on the new root
→ Repeat this process until only one node remains.

Analyzing Heapsort :
Build Max Heap (A) $O(n)$
for i ← Length(A) downto 2
 do exchange A[1] ⟷ A[i] } n−1 times
  Max Heapify (A, 1, i−1) $O(\lg n)$
⇒ $O(n) + (n-1)O(\lg n) = O(n\lg n)$

Heap Maximum (A) $O(1)$
 return A[1]

Heap Extract max (A, n)
 If n < 1
  then error "heap underflow"
 max ← A[1]
 A[1] ← A[n] $O(\lg n)$
 max heapify (A, 1, n−1)
 return max
↳ exchange the root element with the last
↳ Decrease the size of the heap by 1 element
↳ call max heapify on the new root, on a heap of size n−1

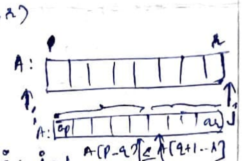
Heap Increase key (A, i, key)
 If key < A[i]
  then error "new key is smaller than current key"
 A[i] ← key
 while i > 1 & A[parent(i)] < A[i]
  do exchange A[i] ⟷ A[parent(i)]
   i ← parent (i) $\boxed{O(\lg n)}$
↳ Increment the key of A[i] to its new value
↳ If the max heap property does not hold anymore. traverse a path toward the root to find the proper place for the newly increased key.

max heap insert (A, key, n)
 heap-size[A] ← n+1 $\boxed{O(\lg n)}$
 A[n+1] ← −∞
 Heap increase key (A, n+1, key)
↳ expand the max heap with a new element whose key is −∞
↳ calls Heap Increase key to set the key of the new node to its correct value and maintain the max heap property.

Quicksort A[p...r] A[p..q] ≤ A[q+1...r]

Quicksort (A, p, r)
 If p < r
  then q ← partition(A, p, r)
   Quicksort (A, p, q)
   Quicksort (A, q+1, r)

Partion (A, p, r)
 x ← A[p]
 i ← p−1
 j ← r+1
 while True
  do repeat j ← j−1
   until A[j] ≤ x
  do repeat i ← i+1
   until A[i] ≥ x $O(n)$
  If i < j
   exchange A[i] ⟷ A[j]
  else return j

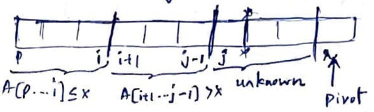


$T(n) = T(q) + T(n-q) + n$

Quicksort worst case :
$T(n) = T(1) + T(n-1) + n \Rightarrow O(n^2)$
Best case : $q = n/2$
$T(n) = 2T(n/2) + n \Rightarrow O(n\log n)$

Randomised partition (A, p, r)
 i ← Random (p, r)
 exchange A[p] ⟷ A[i]
 return partition (A, p, r)
Randomised Quicksort (A, p, r)
 If p < r
 then q ← Randomised partition(A, p, r)
  Randomised Quicksort (A, p, q)
  Randomised Quicksort (A, q+1, r)

Another partitioning : Lomuto's partition



. Pivot x = A[r]
A[p...q−1] smaller than x
A[q+1 ...r] is greater than x
Pivot not included in any of the arrays.

Randomized-Quicksort (A, p, r)
 If p < r
 then q ← Randomized-partion (A, p, r)
  Randomized quicksort (A, p, q−1)
  Randomized Quicksort (A, q+1, r)
n times at most

Partion (A, p, r)
 x ← A[r]
 i ← p−1
 for j ← p to r−1
  do If A[j] ≤ x
   then i ← i+1
    exchange A[i] ⟷ A[j]
 exchange A[i+1] ⟷ A[r]
 return i+1
Total work done $O(nc+X) = O(n+X)$

Expected value $E[X] = \sum_x x \Pr(X=x)$
Indicator random value $I\{A\}$
$\Rightarrow I\{A\} = \begin{cases} 1 & \text{If } A \text{ occurs} \\ 0 & \text{If } A \text{ does not occur} \end{cases}$
Expected value of an Indicator random value $X_A = I\{A\} \Rightarrow E[X_A] = \Pr\{A\}$

Total number of comparisions
$\Rightarrow X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
$\Rightarrow E[X] = E[\;] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$
Pr{z_i is compared to z_j}
$2 \cdot \frac{1}{j-i+1} \quad \frac{1}{j-i+1} = \frac{2}{j-i+1}$
$\Rightarrow E[X] = \sum\sum \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1}$
$< \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \left(\frac{2}{k}\right) (\lg n) \Rightarrow O(n\lg n)$

All our comparison sorts with $\Omega(n\lg n)$
Lowerbound Decision tree $\Omega(n\lg n)$
$n! \le 2^h \Rightarrow \log(n!) \le h$
Stirling's approximation tells $n! > \left(\frac{n}{e}\right)^n$
$\Rightarrow h \ge \lg\left(\frac{n}{e}\right)^n \Rightarrow n\lg n - n\lg e$
$= \Omega(n\lg n)$

Sorting in Linear time :-

counting sort :- no comparisons b/w elements

→ For each element x, find the number of elements $\le x$

→ place x into its correct position in the o/p array.

A → allocate C → Insert in B.

counting sort (A, B, n, k):

for i ← 0 to r
  do C[i] ← 0 } O(r)
for j ← 1 to n
  do C[A[j]] ← C[A[j]]+1 } O(n)
for i ← 1 to r
  do C[i] ← C[i] + C[i-1] } O(r)
for j ← n downto 1
  do B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] − 1 } O(n)

$O(n+r)$

$\approx O(n)$

Radix sort :- represents keys as d-digit numbers in some base-k

eg. key = $x_1 x_2 \cdots x_d$ where $0 \le x_i \le k-1$

key = 15

$key_{10} = 15$, $d = 2$, $k = 10$ where $0 \le x_i \le 9$

$key_2 = 1111$, $d = 4$, $k = 2$ where $0 \le x_i \le 1$

Assumptions :- $d = O(1)$ & $k = O(n)$

→ For a d digit number, sort the least significant digit first using the stable sort algorithm.

→ continue sorting on the next least significant digit, until all digits are sorted

→ Running time $O(d(n+k))$

order statistics :-

→ ith order statistic in a set of n elements is the ith smallest element

→ minimum is 1st order statistic

→ max is nth order statistic

→ median is n/2 th order statistic.

→ n even then 2 medians.

we can find the min & max with less than twice of cost → yes

walk through elements by pairs

→ compare each element in pair to other

→ compare the largest to max & smallest to min

→ Total cost : 3 comparisons per 2 elements $O(3n/2)$

Randomised selection :-

key idea: use partition() from quicksort

But only need to examine one subarray

Saves running time $O(n)$

q = Randomized partition (A, p, r)

[$\le A[q]$ | $\ge A[q]$]   p   k   q   r

Randomized select (A, p, r, i)

if (p == r) then return A[p];
q = Randomized partition (A, p, r)
k = q − p + 1;
if (i == k) then return A[q];
if (i < k) then
  return Randomized select (A, p, q−1, i);
else
  return Randomized select (A, q+1, r, i−k);

Analysing Randomised selection :-

worst case : partition always 0 : n−1

$T(n) = T(n-1) + O(n) = O(n^2)$

Best case : suppose 9:1 partition

$T(n) = T(9n/10) + O(n)$

$= O(n)$

Average case :-

$T(n) \le \frac{1}{n} \sum_{k=0}^{n-1} T(\max(k, n-k-1)) + O(n)$

$\le \frac{2}{n} \sum_{k=n/2}^{n-1} T(k) + O(n)$

$\le \frac{2}{n} \sum_{k=n/2}^{n-1} ck + O(n)$

$= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right) + O(n)$

$\frac{2c}{n} \left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2} \right) + O(n)$

$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}+1\right) + O(n)$

$T(n) \le cn - c - \frac{cn}{4} + \frac{c}{2} + O(n)$

$= cn - \frac{cn}{4} - \frac{c}{2} + O(n)$.

$= cn - \left(\frac{cn}{4} + \frac{c}{2} - O(n)\right)$

$\le cn$

Search problem :- search x such that x = A[i] find index i

Direct addressing :- Assumptions

key values are distinct

Each key is drawn from universe U = {0, 1, ... m−1}

Idea :- store the items in array indexed by keys.

operations :-

Direct address-Search (T, k)
  return T[k]   O(1)
Direct address Insert (T, x)
  T[key[x]] ← x   O(1)
Direct address Delete (T, x)
  T[key[x]] ← NIL   O(1)

Hashtables :-

use function h to compute the slot for each key

store element in slot h(k)

h: U → {0, 1, ... m−1}

collisions :- For a given set (K) of keys

If $|K| \le m$, collisions may or may not happen, depending on the hash function

If $|K| > m$ collisions will definitely happen

chaining :-

put all elements that hash to the same slot into a linked list

→ slot j contains a pointer to the head of the list of all elements that hash to j

Chained-hash-Insert (T, x)   O(1)
  insert x at the head of list T[h(key[x])]
chained hash delete (T, x)
  delete x from the list T[h(key[x])]
Chained hash search (T, k)
  search for element with key k in list T[h(k)]

Running time ∝ length of the list of elements in slot h(k)

Analysis of hashing with chaining

worst case :- All n keys hash to same slot

→ O(n) + time to compute hash function

Average case :- Depends on hash function

Simple uniform hashing assumption

→ Any element is equally likely to hash into any of m slots

Probability of collision $Pr(h(x) = h(y)) = \frac{1}{m}$

length list $T[j] = n_j$ $j = 0 \ldots m-1$

Num of keys in table $n = n_0 + n_1 + \cdots + n_{m-1}$

Avg value of $n_j = E[n_j] = \alpha = n/m$

Load factor of hashtable $\alpha = n/m$

n = number of elements stored in table

m = no of slots in table = no of linked list

α can be <, =, > 1

case 1 unsuccessful search

item not stored in table

takes expected time $O(1+\alpha)$ number of elements examined; computing hash function

case 2 successful search

$O(1+\alpha)$ x_i be the ith element inserted into the hash table,

X_{ij} → indicator random variable

$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E[X_{ij}]\right)$

$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n}\frac{1}{m}\right) = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i)$

$= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) = 1 + \frac{n-1}{2m}$

$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$

Total time $= O(2 + \alpha/2 - \alpha/2n) = O(1+\alpha)$

Division method (hash function)

map a key k to one of the m slots by taking the remainder of k divided by m

h(k) = k mod (m)

multiplication method

multiply key k by constant A where 0<A<1

Extract the fractional part of kA

multiply the fractional part by m

Take the floor of the result

$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m(kA \bmod 1) \rfloor$

universal hashing :-

select a hash function at random, from a designed class of functions at the beginning of execution

Designing a universal class of hash functions

→ choose a prime number p large enough so that every possible key k is in range [0, ... p−1] $Z_p = \{0, 1, \ldots p-1\}$

$Z_p^* = \{1, \ldots p-1\}$

$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$

$\forall a \in Z_p^*$ and $b \in Z_p$

The family of all such hash functions is

$H_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ and } b \in Z_p\}$ univ set

Binary search tree :-

If y is in left subtree of x then key[y] ≤ key[x]

If y is in right subtree of x then key[y] ≥ key[x]

key[leftsubtree(x)] ≤ key[x] ≤ key[rightsubtree(x)]

Inorder : left root right → 2 3 5 5 7 9

Preorder : root left right 5 3 2 5 7 9

postorder : left right root → 2 5 3 9 7 5

Inorder-Tree-walk (x)
  if x ≠ NIL   O(n)
    Inorder Treewalk (left[x])
    print key[x]
    Inorder Tree walk (right[x])

Tree Search (x, k)
  if x = NIL or k = key[x]
    then return x
  if k < key[x]
    then return Tree-Search (left[x], k)
  else return Tree Search (right[x], k)   O(n)

Tree minimum (x)
  while left[x] ≠ NIL   O(h)
    do x ← left[x]
  return x

Tree maximum (x)   O(h)
  while right[x] ≠ NIL
    do x ← right[x]   return x

Successor:- Successor(x) = y, such that key[y] is the smallest key > key[x]

Tree-successor(x):

```
if right[x] ≠ NIL
   then return Tree-minimum(right[x])
y ← p[x]
while y ≠ NIL & x = right[y]
   do x ← y                          O(h)
      y ← p[y]
return y
```

Predecessor:- predecessor(x) = y such that key[y] is the biggest key < key[x]

Tree Insertion:

```
y ← NIL
x ← root[T]
while x ≠ NIL
   do y ← x
      if key[z] < key[x]
         then x ← left[x]
         else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z
   else if key[z] < key[y]
      then left[y] ← z
      else right[y] ← z
```

Idea:- If key[x] < v move to the right child of x else move to the left child of x
→ when x is NIL we found the correct position
→ let y be parent of x
If v < key[y] insert the new node as y's left child
else insert it as y's right child $\boxed{O(h)}$
Beginning at the root go down the tree & maintain:-

Pointer x: trace the downward path (current node)
Pointer y: parent of x ("trailing pointer")

Deletion:-
case 1:- z has no children
Delete z by making the parent of z point to NIL
Case 2:- z has one child
Delete z by making the parent of z point to z's child, instead of z — O(h)
Case 3:- z has 2 children
→ z's successor (y) is minimum node in z's right subtree
→ y has either no children or one right child (but no left child)
→ Delete y from the tree (via case 1 or case 2)
→ Replace z's key and satellite data with y's

Insertion → stable → $O(n^2)$ worst & avg, $O(n)$ almost sorted
Bubble → stable → $O(n^2)$
Radix → stable → $O(nk)$
Counting → stable → $O(n+k)$
Merge → not in place, stable → $O(n \lg n)$
Heap → unstable → $O(n \lg n)$
Quick → unstable → $O(n^2)$ worst, $O(n \lg n)$ best
Selection → unstable → $O(n^2)$

Loop invariant of insertion sort:-
At start of each iteration of the for loop of lines 1-8 the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order
Initialization:- Just before first iteration j = 2: The subarray A[1..j-1] = A[1] is sorted
Maintainance:- while inner loop moves A[j-1], A[j-2], A[j-3] and so on by one position to the right until the proper position for key is found
- At that point the value of key is placed into this position
Termination:- The outer for loop ends when j = n+1 ⇒ j-1 = n
- Replace n with j-1 in the loop invariant ⇒ The subarray A[1...n] consists of the elements originally in A[1...n] in sorted order.

Bubble sort:- At the start of each iteration of the for loop of lines 1-4, the subarray A[1..i-1] consists of the i-1 smallest element in A[1..n] in sorted order. A[i..n] consists of n-i+1 remaining elements in A[1...n]
Initialization:- Initially the subarray A[1..i-1] is empty and trivially this is smallest element of the subarray
maintainance:- After execution of inner loop A[i] will be the smallest element of subarray A[i..n] and in the beginning of the outer loop A[1..i-1] consists of element that are smaller than the elements of A[i..n] in sorted order. So after execution of the outer loop subarray A[1..i] will consist of elements that are smaller than the elements of A[i+1..n] in sorted order
Termination:- The loop terminates when i = A.length. At this point A[1...n] will contain of all elements in sorted order.

Heapsort:- At the start of each iteration of the for loop of lines 2-5 the subarray A[1...n] and A[1..i] is a max heap containing the i smallest elements of A[1..n] sorted. The subarray A[i+1..n] contains the n-i largest elements of A[1..n]
I:- The subarray A[i+1..n] is empty thus the invariant holds
M:- A[1] is the largest element in A[1..i] and it is smaller than the element in A[i+1..n]. when we put it in the i th position, then A[i..n] contain the largest elements sorted. Decreasing the heap size and calling max-Heapify turns A[1..i-1] into a max-heap. Decrementing i sets up the invariant for the next iteration.
T:- After the loop i = 1 This means that A[2...n] is sorted and A[1] is the smallest element in the array which makes array sorted.

Radixsort:- At the beginning of the for loop the array is sorted on the last i-1 digits
In:- The array is trivially sorted on the last 0 digits
M:- Let's assume that the array is sorted on the last i-1 digits. After we sort on the i th digit the array will be sorted on the last i digits. It is obvious that elements with different digits in the i th position are ordered accordingly; In the case of same digit, we still get a correct order, because we're using a stable sort and the elements were already stored on the last i-1 digits
T:- The loop terminates when i = d+1 since the invariant holds, we have the numbers sorted on d digits.